# Ringing $J_s$ refinements

**Jarek Rossignac**
School of Interactive Computing, College of Computing, GVU Center
Georgia Institute of Technology, Atlanta, GA
http://www.gvu.gatech.edu/~jarek

**Scott Schaefer**
Department of Computer Science, 3112 Texas A&M University, College Station, TX 77843-3112
http://faculty.cs.tamu.edu/schaefer

**Abstract**

Both the four-point and the uniform cubic B-spline refinement (i.e. subdivision) schemes double the number of vertices of a closed-loop polygonal curve ${}^{j}P$ and respectively produce sequences of vertices $f_k$ and $b_k$. The $J_s$ refinement proposed here produces vertices $v_k=(1-s)f_k+sb_k$. Iterative applications of $J_s$ yield a family of curves parameterized by s. It includes the four-point curve ($J_0$), the uniform cubic B-spline ($J_{8/8}$), and the quintic B-spline ($J_{12/8}$). Iterating $J_s$ converges to a $C^2$ curve for $0<s<1$, to a $C^3$ curve for $1\leq s<3/2$, and to a $C^4$ curve for $s=3/2$. $J_{3/8}$ tends to reduce the error between consecutive refinements and is useful to reduce popping when switching levels-of-detail in multi-resolution rendering. $J_{4/8}$ produces the Jarek curve, which, in 2D, encloses a surface area that is usually very close to the area enclosed by the original control polygon ${}^{0}P$. We propose model-dependent and model-independent optimizations for these parameter values. As other refinement schemes, the $J_s$ approach extends trivially to open curves, animations, and surfaces. To reduce memory requirements when evaluating the final refined curve, surface, or animation, we introduce a new evaluation technique, called Ringing. It requires a footprint of only 5 points per subdivision level for each curve and does not introduce any redundant calculations.

## 1. Curve refinements

For simplicity, we initially focus on planar, closed loop polygonal curves. Then, we explain how to extend our results to curves in higher-dimensions, to open curves, to motions, and to surfaces. We concentrate on Split&Tweak refinements [Ros04] that insert a new *mid-edge* vertex in the middle of each edge (*Split* operation) and then adjust the position of the old and/or new vertices (*Tweak* operation).

Let ${}^{0}P$ be the initial polygonal control loop and ${}^{k}P$ the loop obtained after k refinement steps (*Split* and *Tweak* pairs). Let ${}^{k}P_j$ be the $j^{th}$ vertex of ${}^{k}P$. We focus on local refinement scheme, where each new vertex of ${}^{k+1}P$ is computed as a linear combination of a set of vertices of ${}^{k}P$. Specifically, we use two stencils: ${}^{k+1}P_{2j} = \alpha\,{}^{k}P_{j-1}+\beta\,{}^{k}P_j+\chi\,{}^{k}P_{j+1}$ for the new position of the old vertices and ${}^{k+1}P_{2j+1} = \delta\,{}^{k}P_{j-1}+\varepsilon\,{}^{k}P_j+\phi\,{}^{k}P_{j+1}+\gamma\,{}^{k}P_{j+2}$ for the position of the mid-edge vertices created by the split. All operations on vertex indices are performed modulo the number of vertices in the loop (Fig. 1 left).

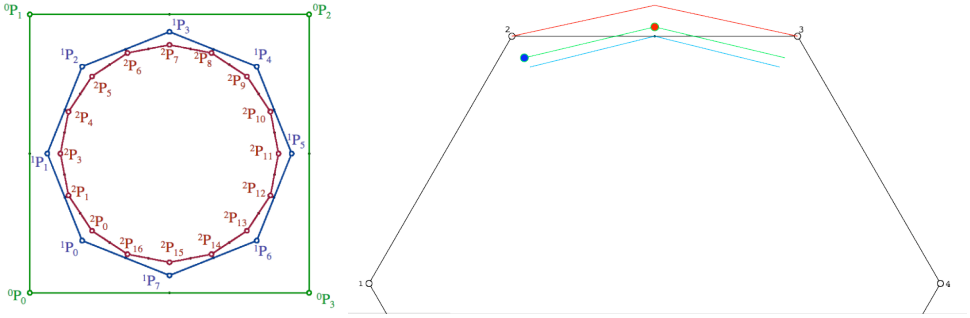

Fig. 1: Results of 3 consecutive refinements (left). The blue point (right) is $\alpha\,{}^{0}P_1+\beta\,{}^{0}P_2+\chi\,{}^{0}P_3$. The red point is $\delta\,{}^{0}P_1+\varepsilon\,{}^{0}P_2+\phi\,{}^{0}P_3+\gamma\,{}^{0}P_4$. Note that these points lie between the corresponding vertices produced by the four-point subdivision (red line) and by the uniform cubic B-spline subdivision (blue line). This particular interpolation corresponds to $J_{0.7}$.

Because we favor symmetric schemes for which the result is independent of the orientation of the loop, we must have $\alpha=\chi$, $\delta=\gamma$, and $\varepsilon=\phi$. To make this subdivision scheme translation invariant, we also enforce that $\alpha+\beta+\chi=1$ and $\delta+\varepsilon+\phi+\gamma=1$. Hence, all seven coefficients may be defined in terms of two parameters $a$ and $b$. We chose $\alpha=\chi=a/8$, $\beta=(8-2a)/8$, $\delta=\gamma=(b-1)/16$, and $\varepsilon=\phi=(9-b)/16$. The corresponding subdivision, ${}^{k+1}P_{2j} = (a\,{}^{k}P_{j-1} + (8-2a)\,{}^{k}P_{j} + a\,{}^{k}P_{j+1})/8$ and ${}^{k+1}P_{2j+1} = ((b-1)\,{}^{k}P_{j-1} + (9-b)\,{}^{k}P_{j} + (9-b)\,{}^{k}P_{j+1} + (b-1)\,{}^{k}P_{j+2})/16$, will be denoted $J_{a,b}$. For simplicity, we use $J_s$ for $J_{s,s}$.

We chose this parameterization so that $J_0$ is the four-point refinement [NDG87, DD89] and $J_1$ is the uniform cubic b-spline refinement [Sab02, LR80] (Fig. 1 left). Also note that $J_{1/2}$ is the Jarek refinement [Ros04], which averages the four-point and cubic b-spline and usually nearly preserves the area enclosed by a 2D curve (Fig. 2). We use ${}^{k}J_s({}^{0}P)$ to denote the result ${}^{k}P$ of applying k $J_s$ refinements to ${}^{0}P$.

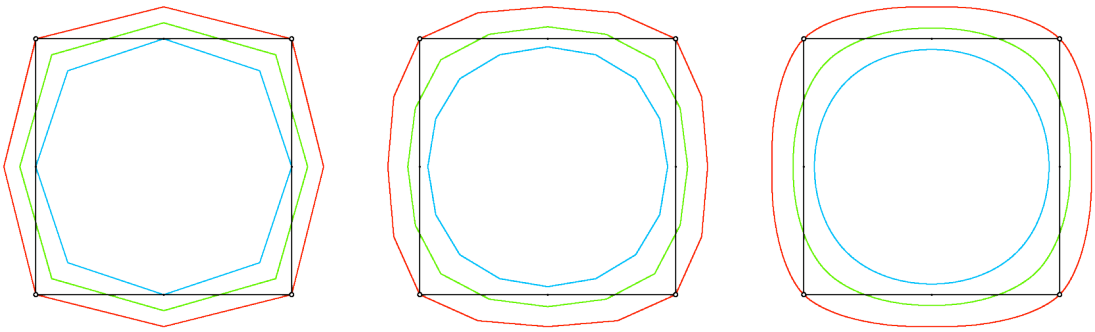

Fig. 2: Results of 1, 2, and 6 refinements of four-point $J_0$ (red), Jarek $J_{1/2}$ (green), cubic B-spline $J_1$ (cyan).

## 2. Previous Work

Several researchers have considered subdivision schemes with a tunable parameter. For example, *splines in tension* [Sch96, Cli74] are a generalization of polynomial splines with a tunable parameter that controls the degree of "tightness" of the curve. These subdivision rules are not linear in the tension parameter. Their smoothness has not been analyzed.

Barsky et al. [BB83] also created a generalization of B-splines called Beta-splines that provide bias and tension controls. Later, Dyn et al. [NDG87] developed an interpolatory four-point subdivision scheme with a tunable parameter that blends between local cubic Lagrange interpolation and linear interpolation. The smoothness of the curves generated with these subdivision scheme depends on the parameter, but is at most $C^1$.

More recently Dyn et al. [DFH05] introduced a subdivision scheme based on local cubic Lagrange interpolation blended with Chaikin's subdivision scheme [Cha74] for $C^1$ B-spline curves. They show that their method generates $C^2$ curves for a large range of parameter values and optimize this parameter to create a subdivision scheme that is as close as possible to being interpolatory.

Despite the fact that several subdivisions schemes with tunable parameters exist, there has been little work on optimizing these parameters to achieve geometric properties such as vertex or mid-edge point interpolation or area preservation. Most subdivision optimization has focused on fitting subdivision surfaces to other data [LLS01, MK04]. Typically these methods are data dependent (their optimization is dependent on the input data) and either use parametric correspondence or attempt to find some geometric correspondence to match the given data. Halstead et al. [HKD93] show how to set up an optimization problem to find the control points for a Catmull-Clark surface that minimizes various energy functionals such as thin-plate energy as well. To eliminate the optimization cost and ensure stability and local control, we aim at choosing optimal parameters for our subdivision scheme independent of any data given.

## 3. Continuity of the $J_s$ family

The $J_s$ refinements generalize the Jarek construction to the whole family of refinement schemes (Fig. 3).

Consider two loops, $P = \{P_0, P_1, \ldots P_k\}$ and $Q = \{Q_0, Q_1, \ldots Q_k\}$. Let $L_s(P,Q)$ produce a new loop $R = \{R_0, R_1, \ldots R_k\}$, where $R_i = (1-s)P_i + sQ_i$. Note that although $J_s({}^{0}P) = L_s(J_0({}^{0}P), J_1({}^{0}P))$, in general, ${}^{k}J_s({}^{0}P) \neq L_s({}^{k}J_0({}^{0}P), {}^{k}J_1({}^{0}P))$. Hence, the curves produced by iterations of $J_s$ refinements are **not linear combinations of the curves produced by iterations of**

**four-point and cubic B-spline schemes**. This observation explains why the limit curves produced by iterative $J_s$ refinements exhibit superior smoothness properties. As the number k of refinements grows, the loop ${}^kP$ converges to a limit curve ${}^*J_s({}^0P)$, which we simply denote as ${}^*J_s$. We show that:

- for $-1.7 \le s < 0$ and $4 \le s \le 5.8$, ${}^*J_s$ is $C^1$,
- for $0 < s \le 1$ and $2.8 < s < 4$, ${}^*J_s$ is $C^2$,
- for $1 < s < 3/2$ and $3/2 < s \le 2.8$, ${}^*J_s$ is $C^3$,
- for $s = 3/2$, ${}^*J_s$ is $C^4$.

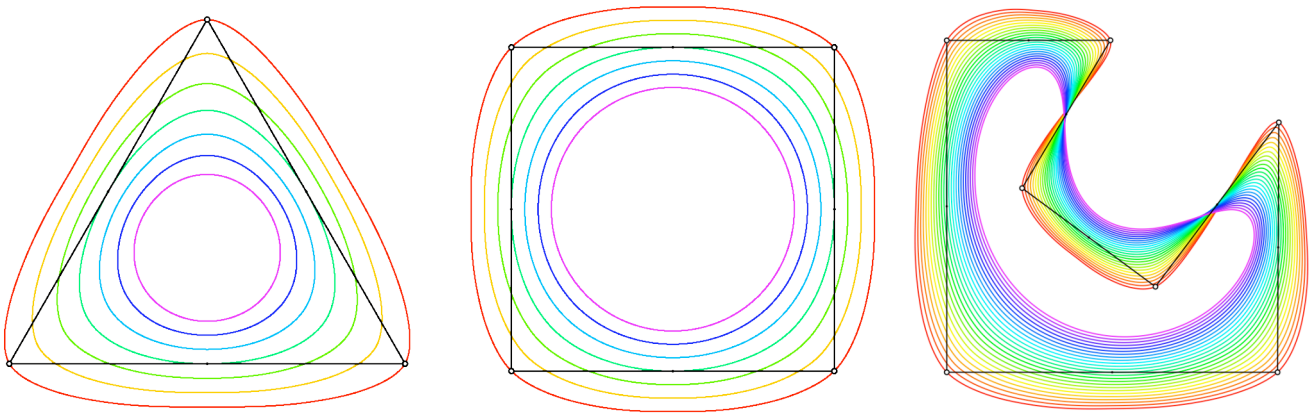

Fig. 3: ${}^5J_0$, ${}^5J_{2/8}$, ${}^5J_{4/8}$, ${}^5J_{6/8}$, ${}^5J_{8/8}$, ${}^5J_{10/8}$, ${}^5J_{12/8}$, colored from red to magenta (left & center). ${}^*J_0$ is the $C^1$ four-point curve (red). ${}^*J_{4/8}$ is the $C^2$ Jarek curve (green). ${}^*J_{8/8}$ is the $C^2$ uniform cubic B-spline curve (cyan). ${}^*J_{12/8}$ is the $C^4$ quintic uniform B-spline curve (magenta). A denser sampling of ${}^*J_s$ curves is also shown (right).

To establish the continuity of the $J_s$ scheme for different values of s, we first consider the necessary conditions for continuity due to Reif [Rei95]. Given the subdivision matrix for $J_s$, if the subdivision scheme produces curves that are $C^m$, then the eigenvalues of its subdivision matrix are of the form 1, (1/2), (1/4), …, $(1/2)^m$, λ, … where $\lambda < (1/2)^m$. The eigenvalues of the subdivision matrix for $J_s$ subdivision are 1, (1/2), (1/4), (1/8), (2–s)/8, (s–1)/16, (s-1)/16, 0, 0. It is easy to verify that $J_s$ subdivision satisfies the **necessary** conditions for $C^1$ continuity when $-2 < s < 6$, $C^2$ continuity when $0 < s < 4$, $C^3$ continuity when $1 < s < 3$, and $C^4$ continuity when $s = 3/2$. Notice that these conditions are only necessary, they are not sufficient.

To determine **sufficient** conditions on the subdivision scheme, we use the Laurent polynomial of the subdivision scheme given by $S(z) = (s-1)/16 + s/8\, z + (9-s)/16\, z^2 + (1-s/4)\, z^3 + (9-s)/16\, z^4 + s/8\, z^5 + (s-1)/16\, z^6$, which encodes the columns of the infinite subdivision matrix in a compact form. The subdivision scheme will generate $C^m$ curves if the infinity norm of the $k^{th}$ power of the subdivision matrix for the $m^{th}$ divided differences is less than 1 for some k [WW02, p. 77]. The columns of this divided difference subdivision matrix are given by $(2^m/(1+z)^{m+1})S(z)$. We can check numerically what range of s satisfies these bounds for different continuity levels. We have verified that $J_s$ subdivision produces curves that are at least $C^1$ for $-1.7 <= s <= 5.8$, $C^2$ for $0 < s < 4$, $C^3$ for $1 < s <= 2.8$ and $C^4$ for $s = 3/2$. In fact, $s = 3/2$ corresponds to uniform quintic b-spline subdivision, which is easily verified by noticing that their Laurant polynomials are identical. Although the sufficient bounds that we were able to verify numerically are slightly more restrictive than the proven necessary bounds, we strongly suspect that the true sufficient bounds extend to match the necessary bounds for continuity in the limit. We were not able to verify this conjecture because the numerical verification is exponential in k and difficult to compute for large values of k.

## 4. Relation with uniform B-splines

Lane and Riesenfeld [LR80] showed that uniform B-spline curves $B_d$ of degree d have a simple subdivision composed of two parts. To subdivide a curve, we first double the control points by inserting mid-edge points. Then we take the dual (replace the vertices by the mid-edge points) d-1 times. These subdivision rules create curves that are $C^{d-1}$.

Our subdivision scheme $J_s$ exactly reproduces the odd degree B-splines $B_3$ and $B_5$ for $s = 1$ and $s = 3/2$. It does not reproduce even degree B-splines exactly though. However, we can optimize our parameter s in a data independent manner to match the basis functions created by $B_2$ and $B_4$ subdivision (Fig. 4).

To perform this optimization, we minimize the difference between the basis function values on a dense uniform grid, but the optimal parameter s will depend on what norm is used to measure the distance between the values. $L_2$ is a popular norm because the resulting optimization problem is polynomial and easy to solve. However, the $L_2$ norm has little to do with how we perceive closeness. Most believe that the $L_\infty$ norm is the best norm because this norm minimizes the worst-case scenario and provides strict error bounds. The disadvantage of the $L_\infty$ norm is that the optimization problem becomes difficult due to the use of non-differentiable functions like Max and Abs. On the other hand, the $L_1$ norm optimizes the average case scenario and will typically perform better in practice than other norms, but this norm does not bound the worst case as the $L_\infty$ norm does. Since we are performing data independent optimization, we can compute the optimal parameter in these different norms even if the computation requires significant effort, because we only need to perform the computation once.

When optimizing our subdivision scheme to match quadratic B-spline subdivision, the $L_1$ and $L_\infty$ norms produce very different values s=.689 and s=.639 respectively. In this case, we choose to use the $L_1$ norm as it will perform better in practice. For quartic B-splines, the two norms are very close to one another and we obtain an optimal value of s=1.27.

Notice that quadratic B-splines are actually $C^1$ curves whereas our $J_{.689}$ subdivision scheme that approximates them actually produces $C^2$ curves. $J_{8/8}$ converges to a cubic B-spline curve $B_3$. $J_{1.27}$ converges to a $C^3$ curve that closely approximates the quartic B-spline curve $B_4$. Finally, $J_{12/8}$ converges to a $C^4$ quintic B-spline curve (Fig. 4).

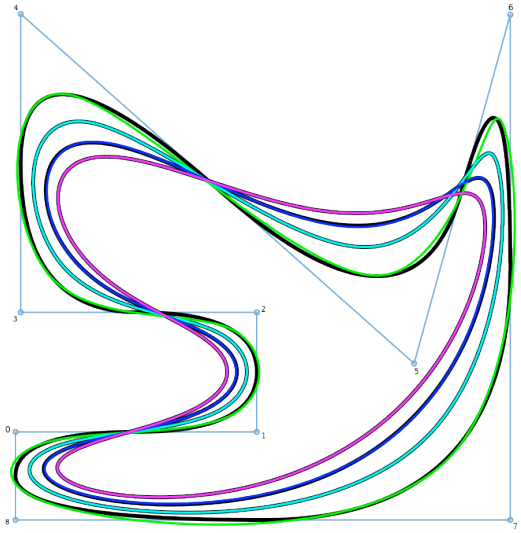


Fig. 4: ${}^*J_{.689}$ (green) approximates $B_2$. ${}^*J_1$ (cyan) is $B_3$. ${}^*J_{1.27}$ (blue) approximates $B_4$. ${}^*J_{1.5}$ (magenta) is $B_5$. To facilitate comparison, the J curves are drawn on top of their thicker black B counterparts.

## 5. Retrofitting

As the value of s increases towards 1.5, the smoothness of our curve increases, but the limit curve drifts farther away from the vertices of the *original control loop* C. The remedy this problem we can perform a simple optimization to obtain a polygon loop ${}^0P$ for which the limit curve ${}^*P$ exactly interpolates the vertices of C. In general, the limit mask for the $J_s$ subdivision is given by the dominant left eigenvector of the subdivision matrix and has the closed-form {(s–1)s, 2s(8–s), 72 + 2(s–9)s, 2s(8–s), (s–1)s}/(12(6+s)) for arbitrary parameter values s. We can solve a global system of equations using a matrix whose rows contain shifts of the limit mask to find control points whose limit curve exactly interpolate the vertices of the control polygon [WW02, p. 182], but this solution may be expensive to calculate for large numbers of control points.

As an alternative, we have implemented an iterative retrofitting (Fig. 5), which can quickly converge to the solution of these equations. We initialize ${}^0P$ with the vertices of C. Then, for each vertex ${}^0P_j$, we compute its limit position ${}^*P_j$ using the limit mask provided above. We then adjust each vertex ${}^0P_j$ to ${}^0P_j+(C_j-{}^*P_j)$. We iterate this process until the difference between ${}^*P_j$ and $C_j$ for all j falls below a desired threshold.

This retrofitting process may be applied to any $J_s$ scheme (Fig. 6). In practice, the iterative solver converges in realtime, making the proposed retrofitting suitable for interactive editing. Unfortunately, we have no proof of

convergence. Furthermore, we show that this method fails to converge for some ranges of s values. We do so by computing the spectral radius (largest absolute eigenvalue) of the infinite matrix (I–L) where I is the identity matrix and L is a matrix whose rows contains shifts of the limit mask. Despite the fact that this matrix is infinite, we can use techniques from block-circulant matrices to write down the infinite set of eigenvalues and bound their norm. If the spectral radius of the matrix (I–L) is greater than or equal to 1, then this iterative method for interpolating the vertices of the control polygon will fail. For our subdivision scheme, we violate this convergence criteria for $s \leq -0.86$ and $2 \leq s$. Therefore, this iterative retrofitting method will not work for these values of s. For $-0.86 < s < 2$ this iterative technique worked well and converged quickly for all of our test cases during interactive curve manipulation.

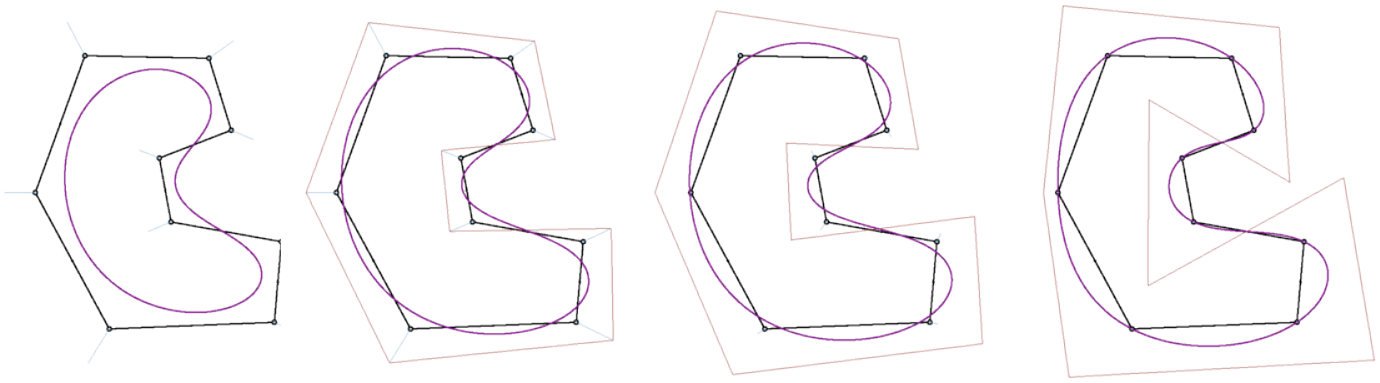

Fig. 5: We compute the vectors $C_j - {}^*P_j$ (left) and apply them (center-left) to adjust ${}^0P_j$. We repeat this process (center-right), quickly converging to a new control polygon (orange), which yields an interpolating curve (right).

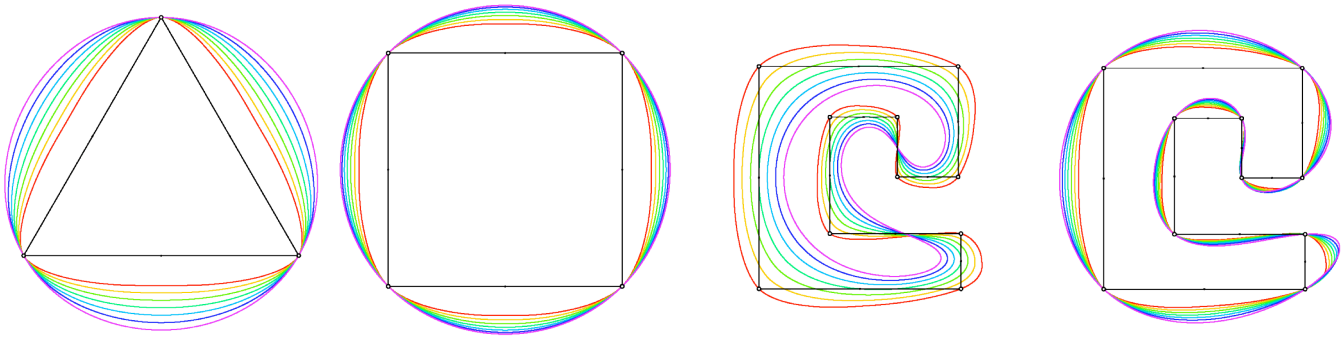

Fig. 6: Retrofitted versions of the refined triangle and square (left) of Fig. 3. The original (center-right) and retrofitted (right) $J_s$ curves for another shape.

Note that the local control property of the $J_s$ refinements is lost when retrofitting; hence each control vertex of C may influence the entire curve ${}^*J_s$, possibly making this approach impractical for precise, local shape editing in some applications. Therefore, we propose below an alternative that makes it possible to retain local control and obtain refined curves that nearly interpolate the vertices of C.

## 6. Vertex-interpolation through mixed schemes

Except for ${}^*J_0$, the ${}^*J_s$ curves are not interpolating. While retrofitting can create an interpolatory curve for a given $J_s$ scheme, the global nature of the solution leads to a loss of local control when editing the curve. To avoid the retrofitting global optimization while bringing the limit curve closer to the control vertices, we propose to combine $J_s$ steps with different values of s. For example, a single anticipation $J_r$ step, with $r = -33/26$, followed by a series of $J_{12/8}$ steps converges to a $C^4$ quintic B-spline curve that nearly interpolates the original vertices (Fig. 7). We obtain exact interpolation of the vertices if we precede the $J_{12/8}$ iterations with an anticipation $J_{a,b}$ step, where $a = -7/4$ and $b = 59/52$, but the final shape is somewhat flattened along the edges (Fig. 8). In both cases, we solve for these parameters by minimizing the difference between the limit masks of the modified curves and the identity mask yielding results independent of a particular shape.

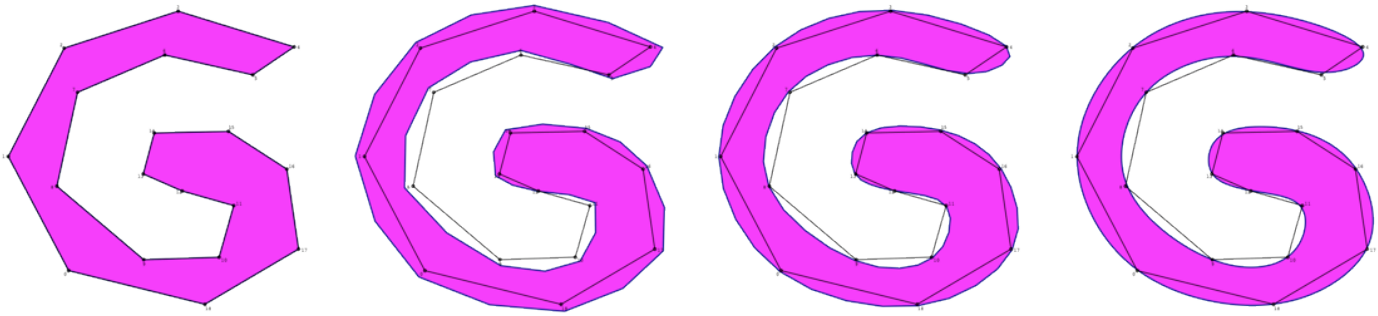

Fig. 7: From left to right: Original. After an "anticipation" step of $J_r$ with r = -33/26. After a subsequent step $J_{12/8}$. Subsequent iterations of $J_{12/8}$ converge to a $C^4$ curve close to the original vertices (right).

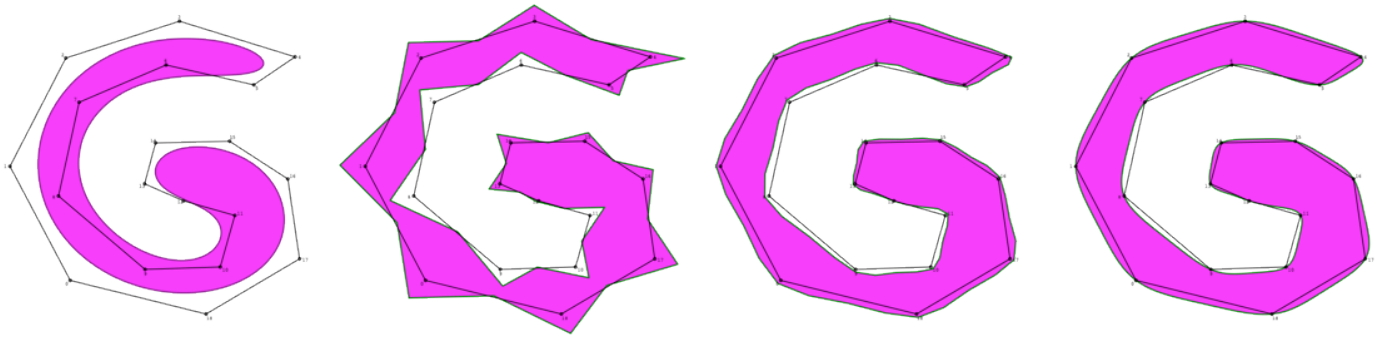

Fig. 8: Left: $*J_{12/8}$. Center-left: $J_{a,b}$, with $a$= -7/4 and $b$= 59/52. Center-right: followed by $J_{12/8}$. Right: followed by several additional $J_{12/8}$.

## 7. Mid-edge interpolation

While the previous section concentrated on interpolating the original control vertices, we note that interpolating mid-edge points is easier and produces a better shaped curve in general. We compare (Fig. 9) several approaches to edge interpolation: the $C^1$ quadratic B-spline $B_2$ curve; the $C^2$ curve produced using a $J_r$ step with r=2/3 followed by a series of $J_1$ steps, the $C^2$ $^*J_s$ with s=0.751, and the $C^4$ curve produced using a $J_r$ step, with r = 29/59, followed by a series of $J_{12/8}$ steps. Note that the first two schemes interpolate the mid-edge points exactly, while the other two only pass very close to them. In each case we derived these parameters by minimizing the difference between the edge limit mask and the midpoint mask in the infinity norm.

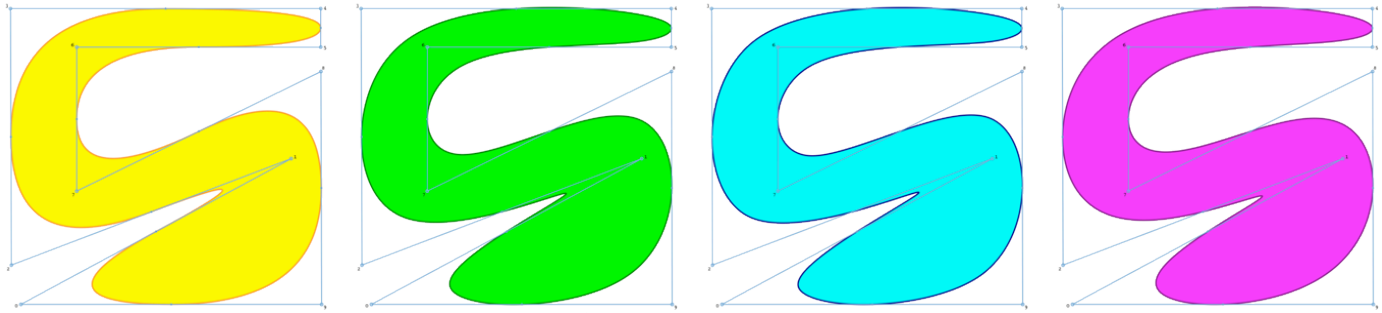

Fig. 9: From left-to-right: $B_2$, $J_{2/3}$ followed by $^*J_1$, $^*J_{0.751}$, $J_{29/59}$ followed by $^*J_{12/8}$.

## 8. Area preservation in 2D

For each control loop, the a, b, and s parameters may be adjusted in a shape-dependent manner through numerical iteration to ensure that the refined curve has the same area as $^0P$. Adjusting s in $*J_s$ will typically produce a $C^2$ curve though this level of smoothness is not guaranteed (Fig. 10a). Adjusting r in $J_r$ followed by $^*J_{1.5}$ will produce a $C^4$ curve (Fig. 10b).

To avoid this shape-dependent optimization, we recommend $^*J_s$ with $s = 0.46415$ (Fig. 10c) or, if a $C^4$ curve is desired, $J_r$, with r = -0.0299, followed by $^*J_{1.5}$ (Fig. 10d). $J_{0.836}$ followed by $^*J_{-0.531}$ yields a $C^1$ curve, with usually a smaller area

error, but exhibits visible kinks due to the lack of higher order continuity (Fig. 10e). These values were derived by minimizing the difference between the exact inner product of the $J_s$ scheme and linear subdivision in the infinity norm. Note that these solutions are independent of the particular control polygon, but do not guarantee that area will be preserved exactly.

If exact area preservation is required with a model-independent solution, we suggest a step of $J_{-0.0053,1.0276}$ followed by ${}^*J_{0.4666}$, which produces a $C^2$ curve that has the same area as the original control polygon, but the curve is noticeably flat along the edges of the control polygon (Fig. 10f).

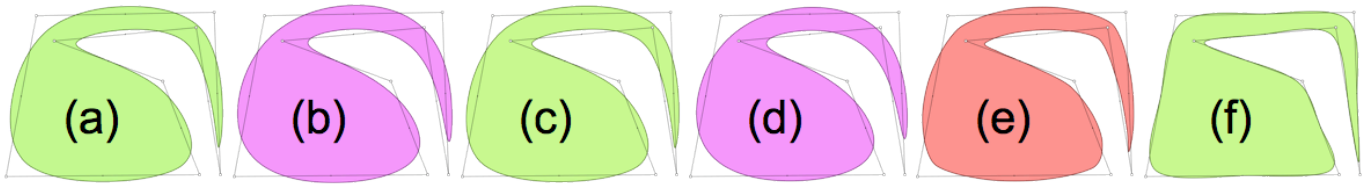


Fig. 10: E is the relative area error. Model-dependent optimizations ${}^*J_{.476}$ (a) or $J_{.050}$ followed by ${}^*J_{1.5}$ (b) both yield E=0. Model-independent optimized approximation through ${}^*J_{.46415}$ (c) yields E=0.00786%, $J_{-0.0299}$ followed by ${}^*J_{1.5}$ (d) yields E=0.03443%, and $J_{0.836}$ followed by ${}^*J_{-0.531}$(e) yields E=0.00014%. $J_{-0.0053,1.0276}$ followed by ${}^*J_{0.4666}$ (f) has exactly the same area as the control polygon independent of what control points are chosen.

## 9. Popping reduction in multi-resolution rendering

To reduce popping when switching between consecutive levels of detail in multi-resolution rendering, we may prefer a smooth curve that reduces the difference between consecutive levels of refinement. Again, we can optimize the s parameter in $J_s$ to match linear subdivision. However, there is a large discrepancy between optimal values in different norms: $L_\infty$ yields s=.152773 whereas $L_1$ yields s=.304763. Although the result may depend on the particular control loop, we believe that the $L_1$ norm performs better for most applications.

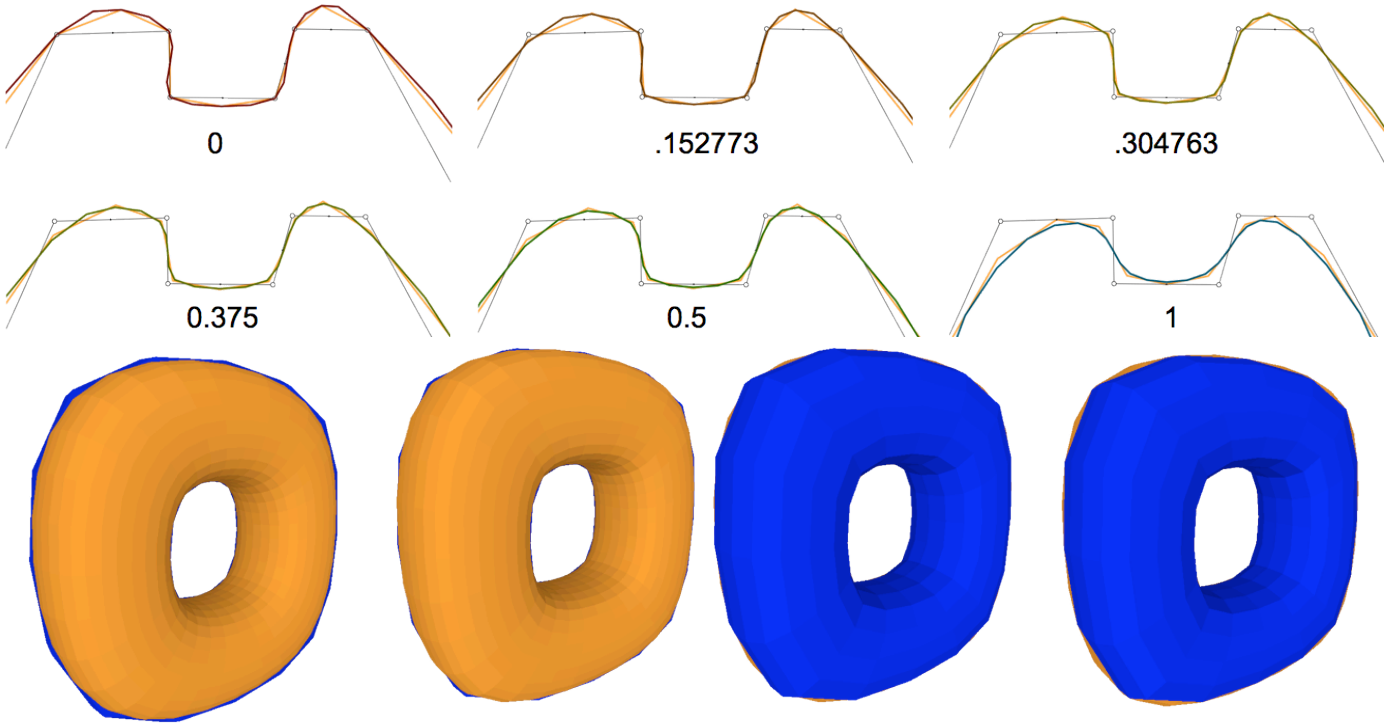


Fig. 11: Top: The previous refinement is shown in orange for various values of s. Bottom: The previous refinement is shown in blue and the images superimposed to show silhouette disparities for $J_0$ (left), $J_{3/8}$ (center), and $J_1$ (right).

## 10.Applications to multi-resolution design

The refinements may be used as in *Hierarchical B-splines* [Fors88] to first define a smooth curve with very few control points and then add small details by editing the position of user-selected vertices at intermediate subdivision levels, before performing subsequent levels of subdivision (Fig. 12).

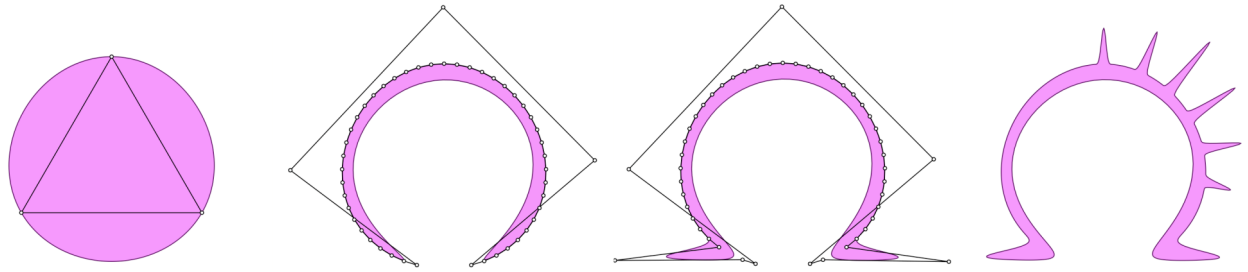

Fig. 12: Only 3 control points were used to create a disk using a $J_{12/8}$ retrofit refinement (left). The 3 bottom vertices of an intermediate refinement were displaced to create a cavity (center-left). 2 other vertices were displaced to bend the ends into an Ω (center-right). 6 vertices of a further refinement were pulled to add 6 spikes before further refinements (right). The final shape was completely specified by only 14 control vertices.

## 11.Open curves

The extension of the $J_s$ refinements to open-loop curves may be performed by inserting 4 *additional control points* between ${}^0P_0$ and ${}^0P_{n-1}$ and by omitting 5 spans (Fig. 13). The additional control points control the behavior of the limit curve near its ends. We choose to make the limit curve interpolate (in position and direction) both ends of the original control polygon: that is to start at ${}^0P_0$ with a tangent along ${}^0P_1-{}^0P_0$ and to end at ${}^0P_{n-1}$ with a tangent along ${}^0P_{n-2}-{}^0P_{n-1}$.

Using the limit mask from Section 5 and the tangent mask {1-s,2(s-4),0,-2(s-4),-(1-s)}/12 derived from the left eigenvector of the subdivision matrix corresponding to ½, we solve a simple set of equations for these two additional control points to enforce these specified conditions. The solution is to add two control points ${}^0P_{-1} = (9-s)/4\ {}^0P_0 + (s-3)/2\ {}^0P_1 + (1-s)/4\ {}^0P_2$ and ${}^0P_{-2} = (12-s)/2\ {}^0P_0 + (s-8)\ {}^0P_1 + (6-s)/2\ {}^0P_2$ to the curve. The masks for the opposite end of the curve are identical.

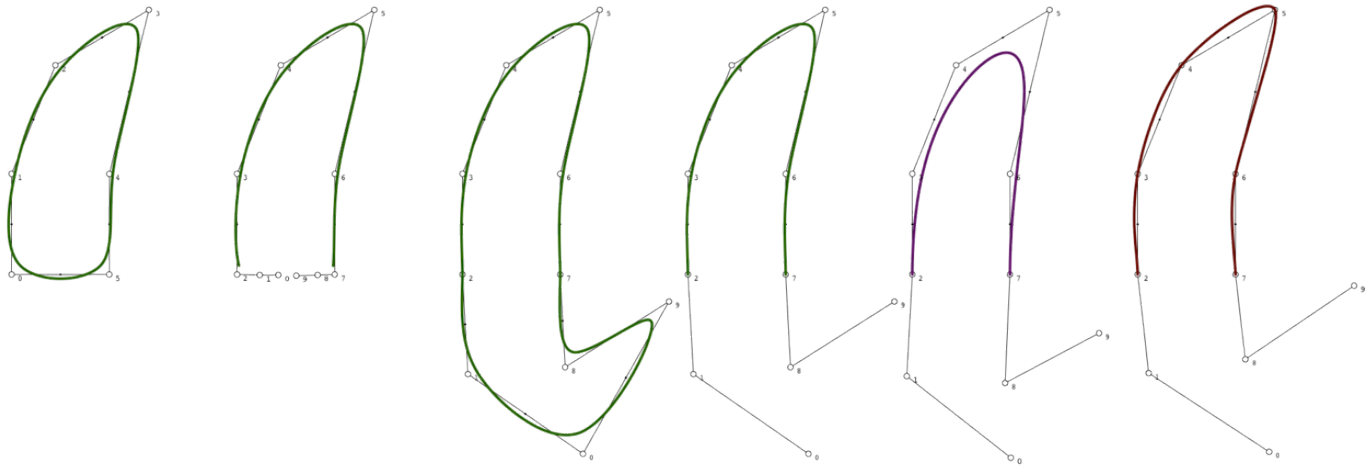

Fig. 13: From left, ${}^*J_{0.5}$ closed loop; With 4 vertices add between ${}^0P_5$ and ${}^0P_0$ and 5 spans removed; As a closed loop with the 4 new vertices adjusted; With 5 spans removed; With different adjustments for ${}^*J_0$ ; And for ${}^*J_{1.5}$.

## 12.Space-conscious refinement (*Ringing*)

A naïve Split&Tweak-like implementation [Ros04] of the $J_s$ refinements proposed here is trivial, but requires storing all the points of the final curve (or at least on the penultimate curve). Although the required storage is rarely a problem when rendering a single curve, it may be an issue when using the $J_s$ refinements to display surfaces or animations with a large numbers of recursions, or when the refinements are performed on graphics hardware with limited on-chip

memory. Therefore, we propose a novel approach for producing one-by-one the series of consecutive points on the final curve without having to store the intermediate levels of subdivision and without having to perform any redundant computation.

The proposed *Ringing* approach uses a **ring** of 5 points per level L of subdivision. At any given moment during the curve evaluation or rendering, ring $r_k$ contains 5 consecutive points of ${}^kP$. During curve evaluation or rendering, $r_k$ slides along ${}^kP$, one vertex at a time. Each rings stores its point in an array of 5 sots. To avoid shifting points we advance the index to the last vertex (the one that will be replaced next) using modulo 5.

The rings are synchronized, so that $r_{k+1}$ advances twice faster than $r_k$. The top ring, $r_0$, obtains its next point as the next point along ${}^0P$. Each other ring $r_{k+1}$, for $k\geq 0$, computes its next point from $r_k$ alternating the two $J_s$ masks introduced in Section 1: ${}^{k+1}P_{2j} = (a\,{}^kP_{j-1} + (8-2a)\,{}^kP_j + a\,{}^kP_{j+1})/8$ for each even point and ${}^{k+1}P_{2j+1} = ((b-1)\,{}^kP_{j-1} + (9-b)\,{}^kP_j + (9-b)\,{}^kP_{j+1} + (b-1)\,{}^kP_{j+2})/16$ for each odd point.

At initialization, the top ring $r_0$ is loaded with the first 5 control points of $P^0$. The points of the other rings, $r_1$, $r_2$,… $r_L$, are derived recursively using these two refinement formulae.

Then, we advance the bottom ring $r_L$ one step at a time, sliding at each step its 5 points by one vertex along the final curve. For every 2 steps of $r_k$ the parent ring $r_{k-1}$ makes one step. To advance $r_0$, we load it with the next control point on the curve. The steps are shown (Fig. 14). Details of our implementation are provided in the Appendix.

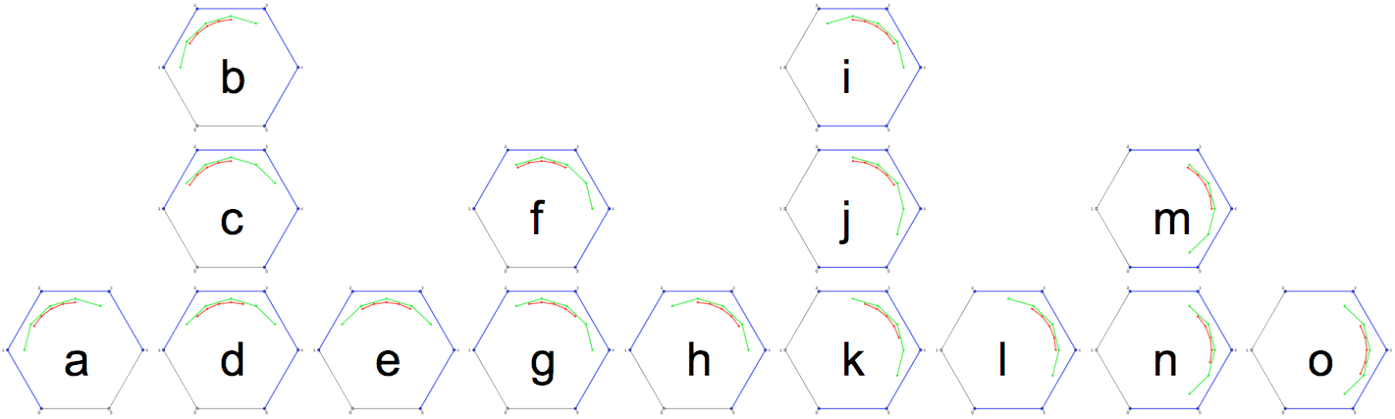


**Fig. 14**: For clarity, we set the number of subdivisions to L=2, we show $r_0$ (blue), $r_1$ (green), and $r_2$ (red). (a) During initialization, $r_0$ is loaded with the first 5 control vertices; points of $r_1$ are derived from points in $r_0$; and points of $r_2$ are derived from points in $r_1$ using functions, $b_1$, $f_{12}$, $b_2$, $f_{23}$, $b_3$, which are provided in the Appendix. (b) We advance $r_0$ by pushing the 6th control vertex in the FIFO of $r_0$. (c) We advance $r_1$ by computing its new vertex from the last 4 vertices of $r_0$ (call to function $f_{23}$). (d) We advance $r_0$ by computing its new vertex from the last 4 vertices of $r_1$ (call to $f_{23}$). Note that the 5 points in the FIFO of $r_2$ have moved by 1 vertex along the final refined curve. (e) We advance $r_0$ again by computing its new vertex from the last 3 vertices of $r_1$ (call to function $b_3$). (f) We advance $r_1$ by computing its new vertex from the last 3 vertices of $r_0$ (call to $b_3$). (g) We advance $r_0$ (call $f_{23}$). (e) We advance $r_0$ (call to $b_3$).

## 13.Extensions to surfaces and animations

Although for clarity we have used 2D curves for illustration, the proposed approach may be extended to refine curves in higher dimensions or curves with properties.

The ringing approach described above, generates points on the subdivided curve one by one. Hence it may trivially adapted to define the trajectory of a moving point or of the center of a moving object and to animate it.

Replacing each control points by a different trajectory defines an animated curve that deforms through time where, at each step of the animation, we advance each control point by one step along its trajectory and then draw the subdivision curve they define. This animation approach uses one set of *motion-rings* per moving control point and one set of *display-rings* for drawing the current curve at a given time. For example, we take the third point (point C) of each final motion-ring and use them as a control polygon for driving the display-ring and drawing the current curve.

To produce surfaces, we proceed as suggested above, but use a second set of *display-rings*, driven using the fourth point (say point D) on each final motion-ring. We drive the two *display-rings* simultaneously to produce a string of quads along the surface (Fig. 15).

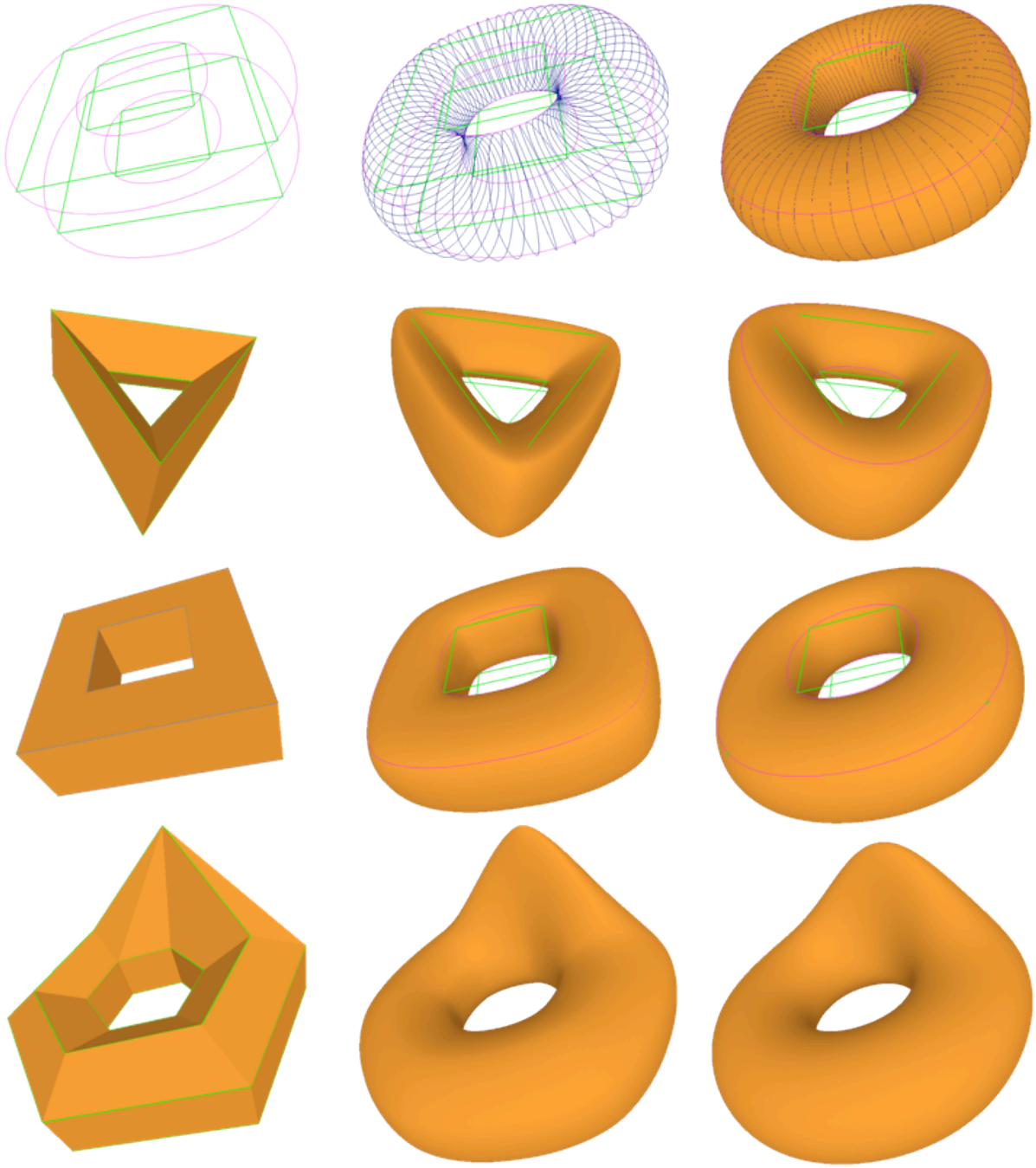

**Fig. 15**: Top: A torus-like surface defined 4 green control curves (trajectories) with 4 control vertices each. The refined versions of these curves are shown in magenta (left). Each set of 4 corresponding moving control points, one on each trajectory, defines a blue transversal curve (center). Triangle strips formed by pairs of consecutive transversal curves are shaded (right). Below: The control polyhedron (left), $*J_1$ (center), and $J_r\ {}^5J_{12/8}$ (right) are shown for three control meshes. The rendering was performed using a footprint of respectively 5, 6, and 8 rings of 5 points each.

To produce surfaces with borders, we add automatically adjusted endpoints (as explained in Section 11) to each motion-ring and to the display-rings and treat them as open-loop curves (Fig. 16).

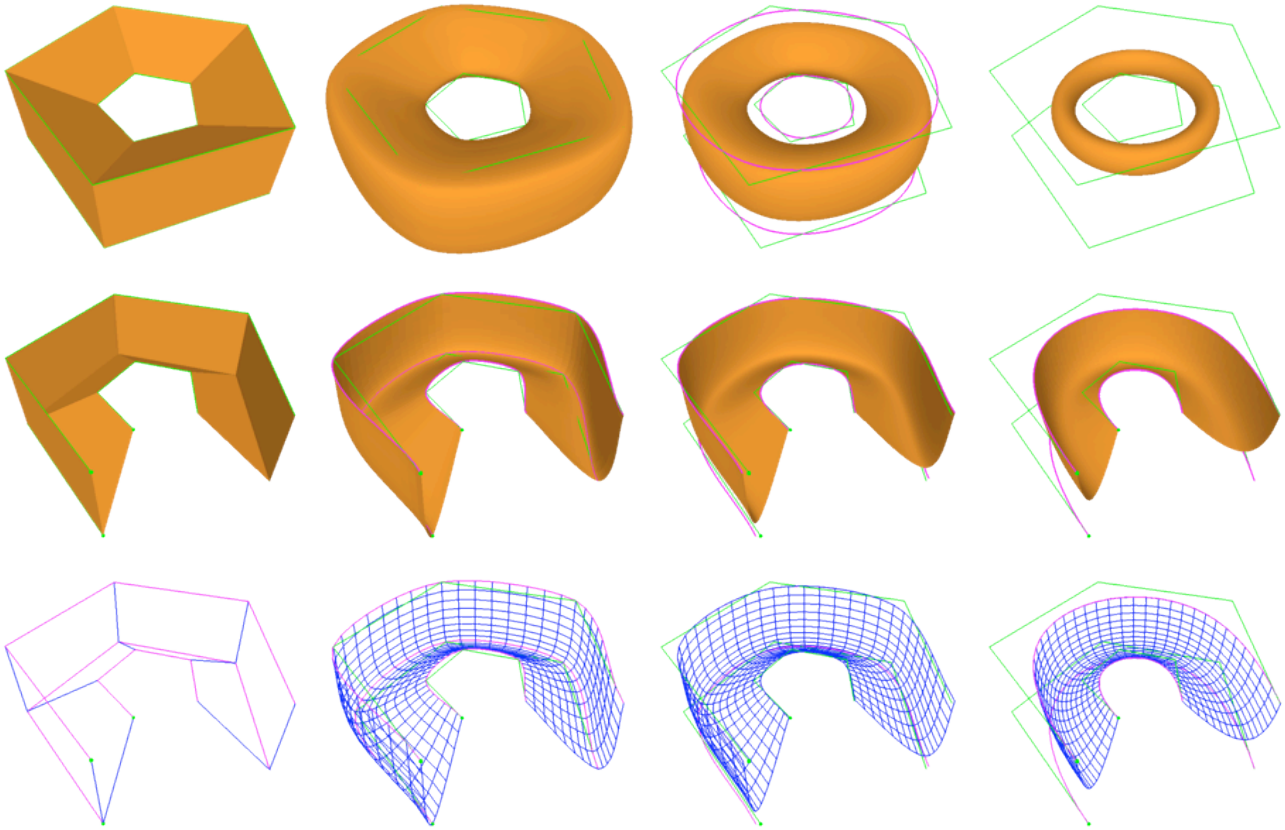

**Fig. 16**: A surface defined by 3 curves of 5 points each. From left to right: control polyhedron, four-point, Jarek, and the quintic B-spline. Closed surfaces (top row), open surfaces with one border (middle), and quads drawn (bottom).

## 14.Conclusions

We have introduced the $J_s$ family of polygonal refinements. It includes the four-point and odd-degree uniform B-spline subdivision and close approximations of even-degree B-spline subdivision.

The s parameter may be optimized for any given control curve—or independently of it if one wishes to preserve local control—to minimize area change, popping in multi-resolution rendering, or distance between the limit curve and the control polygon, its vertices, or its mid-edge points. By limiting this optimization to the first refinement, we trade these fidelity measures for increased smoothness, yielding for example a $C^4$ curve.

These refinements may be used in a variety of 2D and 3D applications for the design and rendering of curves, surfaces, and animations.

We provide the code for a simple *Ringing* implementation of the evaluation of $J_s$ curves, animations, and surfaces that does not require storing the intermediate levels of refinement. Ringing stores 5 points per level of subdivision of a curve and 5(n+2) points per level of subdivision of a surface defined by a control polygon with n rows of control points.

A closed-loop (circle-like) curve may be defined by 3 control points and a genus-one (torus-like) surface may be defined by a 3×3 control grid. If an open curve is preferred, two control points are added at each end. We provide simple expressions (parameterized by s) for computing their positions to ensure that the curve interpolates the ends of the control polygon in position and direction. This approach extends trivially to surface patches with two borders (cylinder-topology) or one border (rectangle with disk-topology).

## 15.Acknowledgement

We thank Malcom Sabin for originally suggesting that the Jarek subdivision may be $C^2$ and hence inspiring these more recent developments.

# 16.Bibliography


[BB83] B. Barsky and J. Beatty, "Local control of bias and tension in beta-splines," In SIGGRAPH '83: Proceedings of the 10th annual conference on computer graphics and interactive techniques (1983), ACM Press, pp. 193-218, 1983.

[Cha74] G. Chaikin, "An algorithm for high speed curve generation," Computer Graphics and Image Processing, vol. 3, pp. 346-349, 1974.

[Cli74] A. Cline, "Scalar- and planar-valued curve fitting using splines under tension," Commun. ACM, vol. 17, pp. 218-220, 1974.

[DD89] G. Deslauriers and S. Dubuc, "Symmetric iterative interpolation processes," Constructive Approximation, vol. 5, pp. 49–68, 1989.

[DFH] N. Dyn, M. Floater, K. Hormann, "A $C^2$ four-point subdivision scheme with fourth order accuracy and its extensions ," In Mathematical Methods for Curves and Surfaces: Tromso 2004, Modern Methods in Mathematics, pp. 145–156, 2005.

[DLG90] N. Dyn, D. Levin, and J. A. Gregory, "A butterfly subdivision scheme for surface interpolation with tension control," ACM Transactions on Graphics, vol. 9, no. 2, pp. 160–169, 1990.

[Duf86] T. Duff: Splines in animation and modeling. In SIGGRAPH '86 Course Notes on State of the Art Image Synthesis (1986), ACM Press.

[Fors88] D. Forsey and R. Bartels, Hierarchical B-Spline Refinement. Proceedings of SIGGRAPH '88, p.205.

[HKD93] M. Halstead, M. Kass, T. DeRose, "Efficient, Fair Interpolation using Catmull-Clark Surfaces," Computer Graphics, vol. 27, pp. 35-44, 1993.

[Kob96] L. Kobbelt: Interpolatory subdivision on open quadrilateral nets with arbitrary topology. Comput. Graph. Forum 15, 3 (1996), 409–420.

[LLS01] N. Litke, A. Levin, P. Schroder, "Fitting subdivision surfaces," In VIS '01, pp. 319-324, 2001.

[Loop94] C. Loop, "Smooth Spline Surfaces over Irregular Meshes", Computer Graphics, Vol. 28, Number Annual Conference Series, pp. 303-310, July 1994.

[LR80] J. Lane and R. Riesenfeld. "A theoretical development for the computer generation and display of piecewise polynomial surfaces," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 2, pp. 25-46, 1980.

[MK04] M. Marinov and L. Kobbelt. "Optimization techniques for approximation with subdivision surfaces," In Symp. Solid Modeling and Applications, pp. 113-122, 2004.

[NDG87] N. Dyn, J. A. Gregory, D. Levin: A four-point interpolatory subdivision scheme for curve design. Computer Aided Design 4 (1987), 257–268.

[Proc07] http://processing.org/

[Rei95] U. Reif. "A unified approach to subdivision algorithms near extraordinary vertices," Computer Aided Geometric Design, vol. 12, pp. 153-174, 1995.

[Ring07] http://www.gvu.gatech.edu/~jarek/demos/ring

[Ros04] J. Rossignac: Education-driven research in CAD. Computer Aided Design 36, 14 (2004), 1461–1469.

[Sab02] M. Sabin: Subdivision surfaces. In The Handbook of Computer-Aided Geometric Design, G. Farin, J. H., Kim M. S., (Eds.). 2002, ch. 12, pp. 309–327.

[Sch66] D. Schweikert, "An interpolation curve using a spline in tension," Journal of Mathematics and Physics, vol. 45, pp. 312-317, 1966.

[SD92] K. Shoemake, T. Duff: Matrix animation and polar decomposition. In Proceedings of the conference on Graphics interface '92 (1992), Morgan Kaufmann Publishers Inc., pp. 258–264.

[Sho85] K. Shoemake: Animating rotation with quaternion curves. In SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques (1985), ACM Press, pp. 245–254.

[WD05] J. Wallner, N. dcyn: Convergence and $C^1$ analysis of subdivision schemes on manifolds by proximity. Computer Aided Design 22, 7 (2005), 593–622.

[WJ93] W. Wang, B. Joe: Orientation interpolation in quaternion space using spherical biarcs. In Graphics Interface (1993), 24–32.

[WP06] J. Wallner, H. Pottmann: Intrinsic subdivision with smooth limits for graphics and animation. ACM Trans. Graph. 25, 2 (2006), 356–374.

[WW02] J. Warren and H. Weimer, "Subdivision methods for geometric design," San Francisco: Morgan Kaufmann, 2002.

[ZSS96] D. Zorin, P. Schroeder, and W. Sweldens, "Interpolating subdivision for meshes with arbitrary topology," Computer Graphics, vol. 30, pp. 189–192, 1996.

# 17.Appendix 1: Ringing code

The *Ringing* code included here is written in Processing [Proc07]. An online application with the entire source code may be found at [Ring07]. Let P be a polyloop. We initialize its first ring R[0] with **P.loadRing();** and the other rings R[1], R[2],… R[L] with **P.deriveRings();**. Then, we advance R[L] along the curve be repeating **P.next();** The **stepper.next();** call to an object of the class **Stepper** returns the highest index of the rings must be advanced at each step.

```
class Stepper {                                                          // stepper for the rings
 boolean [] B = new boolean [10];                                        // Boolean flags and number of recursions
 int d=0;
 Stepper (int pd) {d=pd; this.reset();};
 void reset() {for(int i=0; i<d; i++) B[i]=true; d=rec;}
 int next() {int c=0; while(B[c]&&(c<d)) {B[c]=false; c++;}; B[c]=true; return(c);}   // returns ID of ring that should do a b3 step
 }
int n(int c) {return((c+1)%5);}  int p(int c) {return((c+4)%5);}         // next and previous in ring
pt l(pt A, float s, pt B) {return(new pt(A.x+s*(B.x-A.x),A.y+s*(B.y-A.y),A.z+s*(B.z-A.z)));};   // linear interpolation
pt b(pt A, pt B, pt C, float s) {return( l(l(B,s/4.,A),0.5,l(B,s/4.,C)));};   // tucks in a vertex towards its neighbors
pt f(pt A, pt B, pt C, pt D, float s) {return( l(l(A,1.+(1.-s)/8.,B) ,0.5,l(D,1.+(1.-s)/8.,C)));};   // bulges out a mid-edge point
class ring {                                                             // ring for traversing refined curves
   pt[] P = new pt[5];                                                   // a FIFO of 5 points {A,B,C,D,E}
   int c=2;                                                              // index of middle point C  (it is rotated at each step to avoid copying points)
   ring () {for (int i=0; i<5; i++) P[i]=new pt(0,0);};
 void push (pt F) {c=n(c); P[n(n(c))]=F.make();}                         // loads new point and advances index
 void reset() {c=2;};
 pt pt() {return(P[c].make());}
 pt b1(float s)  {pt bb = b( P[p(p(c))],P[p(c)],P[c],            s); return(bb);}   // b for second vertex
 pt f12(float s) {pt bb = f(P[p(p(c))], P[p(c)],P[c],    P[n(c)],  s); return(bb);}   // f for second mid-edge
 pt b2(float s)  {pt bb = b(P[p(c)],   P[c],  P[n(c)],           s); return(bb);}   // b for third vertex
 pt f23(float s) {pt bb = f(P[p(c)],   P[c],  P[n(c)],  P[n(n(c))],s); return(bb);}   // f for fourth mid-edge
 pt b3(float s)  {pt bb = b(P[c],     P[n(c)],P[n(n(c))],        s); return(bb);}   // b for fifth vertex
 void derive(ring Q, float a, float b) {c=2; P[0]=Q.b1(a); P[1]=Q.f12(b); P[2]=Q.b2(a); P[3]=Q.f23(b); P[4]=Q.b3(a);}   // makes ring from parent ring
 void show() {beginShape(); int b=p(p(c)); for(int i=0; i<5; i++) {P[b].vert(); b=n(b);}; endShape(); for(int i=0; i<5; i++) P[i].show(6);} // show ring
 }
class Polyloop {                                                         // class of polyloops (closed loop polygon)
 int vn = 5, cap=5000;                                                   // number of control vertices and the cap on vn
 pt[] P = new pt [cap];                                                  // control points
 ring [] R = new ring[7];                                                // 7 rings
 int rc;                                                                 // counter showing the next control point to load in the top rig
 Stepper stepper = new Stepper(rec);                                     // stepper for knowing which ring to advance
 Polyloop () {                                                           // creates empty poly
   vn=0; for (int i=0; i<cap; i++) P[i]=new pt(0,0); for(int i=0; i<7; i++) R[i] = new ring(); }

 void pushRing() {R[0].push(P[rc]); rc=this.in(rc);}                     // pushes the next control point to the top ring
 void loadRing() {stepper.reset(); R[0].reset(); rc=0; for(int i=0; i<5; i++) {R[0].push(P[rc]); rc=this.in(rc); }; }   // pushes first 5 points to top ring
 void deriveRings() {R[0].reset(); for (int r=1; r<=rec; r++) {float a=gs, b=gs;  if (r==1) {a=ga; b=gb;}; R[r].derive(R[r-1],a,b);}; }  // derive all other rings
 void showRing(int r) {R[r].show();}                                     // shows 5 points of ring (for demonstration only)
 void f(int r) {float a=gs, b=gs;  if (r==1) {a=ga; b=gb;}; R[r].push(R[r-1].f23(b));}   // pushes ring r with the f23 of parent ring
 void b(int r) {float a=gs, b=gs;  if (r==1) {a=ga; b=gb;}; R[r].push(R[r-1].b3(a));}   // pushes ring r with the b3 of parent ring
 pt next() {int level=rec-stepper.next(); if(level==0) this.pushRing(); else this.b(level); for (int r=level+1; r<=rec; r++) this.f(r); return(R[rec].pt());} // advances last ring by one point along curve
 void showRefined() {this.loadRing(); this.deriveRings(); beginShape(); for (int j=0; j<vn*int(pow(2,rec)); j++) this.next().vert(); endShape(CLOSE); }  // displays the curve
```